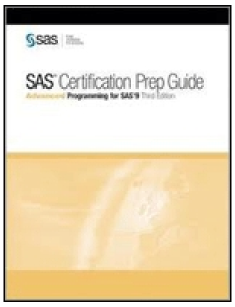


SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

by SAS Institute

Chapter 21: Controlling Data Storage Space

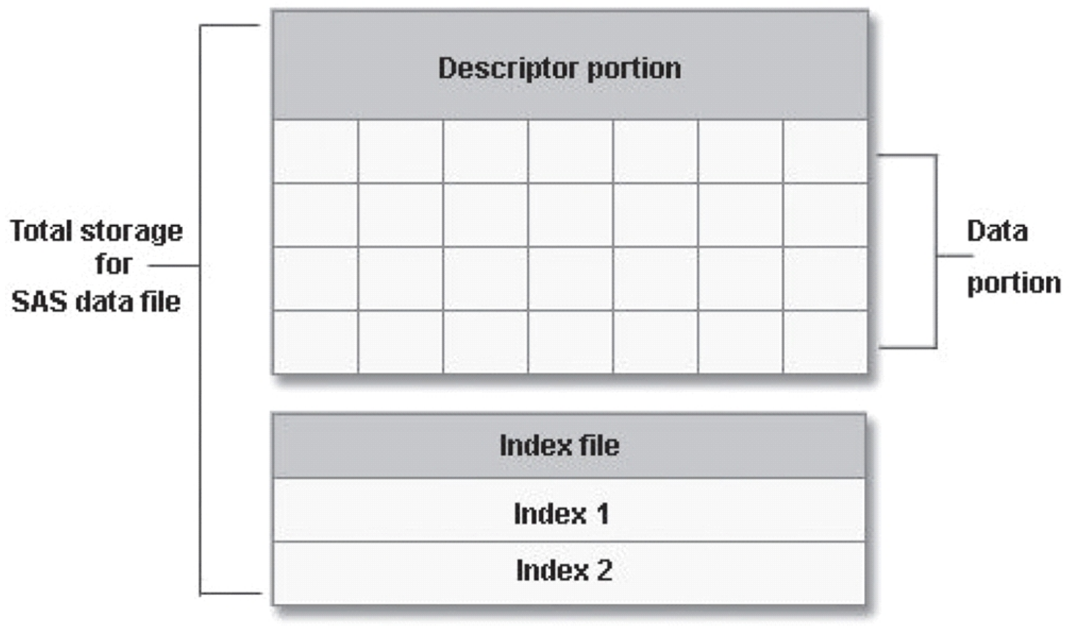
Overview

Introduction

In many computing environments, data storage space is a limited resource. Therefore, it might be more important for you to conserve data storage space than to conserve other resources.

When you store your data in a SAS data file, you use the sum of the data storage space that is required for the following:

- the descriptor portion
- the observations
- any storage overhead
- any associated indexes.



In this chapter you learn to use a variety of techniques for minimizing the amount of space that your SAS data files occupy.

Objectives

In this chapter, you learn to

- describe how SAS stores character variables
- determine how to reduce the length of character variables and how to expand the values
- describe how SAS stores numeric variables
- determine how to safely reduce the space that is required for storing numeric variables in SAS data sets
- define the structure of a compressed SAS data file
- create a compressed SAS data file
- examine the advantages and disadvantages of compression
- describe the difference between a SAS data file and a SAS data view

- examine the advantages and disadvantages of DATA step views.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- Part 1: SQL Processing with SAS
    - "Performing Queries Using PROC SQL" on page 4
    - "Performing Advanced Queries Using PROC SQL" on page 29
    - "Combining Tables Horizontally Using PROC SQL" on page 86
    - "Combining Tables Vertically Using PROC SQL" on page 132
    - "Creating and Managing Tables Using PROC SQL" on page 175
    - "Creating and Managing Indexes Using PROC SQL" on page 238
    - "Creating and Managing Views Using PROC SQL" on page 260
    - "Managing Processing Using PROC SQL" on page 278
- Part 3: Advanced SAS Programming Techniques
    - "Creating Samples and Indexes" on page 470
    - "Combining Data Vertically" on page 502
    - "Combining Data Horizontally" on page 534
    - "Using Lookup Tables to Match Data" on page 580
    - "Formatting Data" on page 626
    - "Modifying SAS Data Sets and Tracking Changes" on page 656
- Part 4: Optimizing SAS Programs
    - "Introduction to Efficient SAS Programming" on page 701
    - "Controlling Memory Usage" on page 711

Reducing Data Storage Space for Character Variables

One way to reduce the amount of data storage space that you need is to reduce the length of character data, thereby eliminating wasted space. Before discussing how to reduce the length of a character variable, consider how SAS assigns lengths to character variables.

How SAS Assigns Lengths to Character Variables

SAS character variables store data as one character per byte. A SAS character variable can be from 1 to 32,767 bytes in length.

The first reference to a variable in the DATA step defines it in the program data vector and in the descriptor portion of the data set. Unless otherwise defined, the first value that is specified for a SAS character variable determines the variable's length. For example, if the length of a character variable called **`Name`** has not been defined and if the first value that is specified for it is *Smith*, the length of **`Name`** is set to 5. Then, if the next value specified for **`Name`** is *Johnson*, the value is stored as *Johns* in the data set. Similarly, if the first value specified for **`Town`** is *Williamsburg*, the length of **`Town`** is set to 12. If the next value for **`Town`** is specified as *Cary*, the length is still 12, and the value is padded with blanks to fill the extra space.

Keep in mind that the first reference to a variable might not be an assignment statement. If SAS cannot determine a length for a character variable when the variable is created in the program data vector, SAS assigns a default length of 8 bytes to the variable.

Reducing the Length of Character Data with the LENGTH Statement

You can use a LENGTH statement to control the length of character variables. It is useful to reduce the length of a character variable with a LENGTH statement when you have a large data set that contains many character variables.

General form, LENGTH statement for character variables:

LENGTH *variable(s) $ length;*

where

variable(s)

specifies the name of one or more SAS variables, separated by spaces.

length

is an integer from 1 to 32,767 that specifies the length of the variable(s).

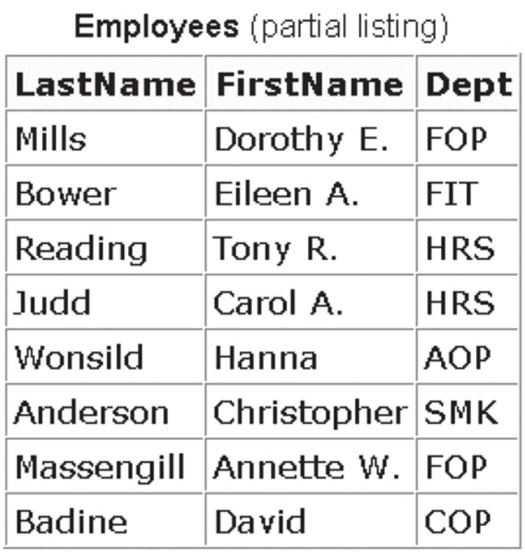
Note Make sure the LENGTH statement appears before any other reference to the variable in the DATA step. If the variable has been created by another statement, then a later use of the LENGTH statement will not change its size.

Other Techniques

There are other techniques that you can use to reduce the length of your character data— especially if your data set has repeated values.

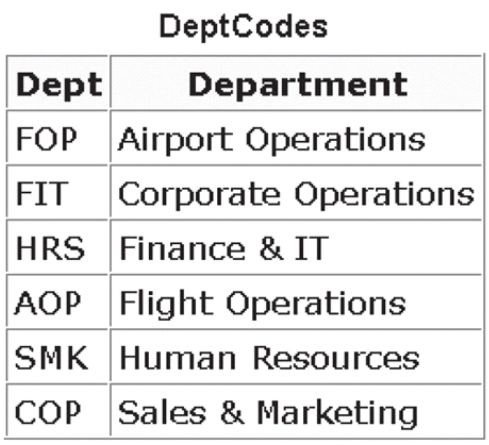
For example, suppose you have a data set that records employee names along with the employees' departments. Instead of recording the complete department name in the data set, you could assign a code to each department and record the code in your data set instead. You could record the complete department name along with the corresponding code in a separate data set, where you would have to record each full-length value only once. Then you could use a table lookup operation to convert the code to the department name for reporting purposes. This is called normalizing the data.

The tables below represent two data sets. *Employees* records the employee names and department codes for all employees. *DeptCodes* records the department name and department code, and is shown in its entirety. If the *Employees* data set contains several hundred observations, then using department codes instead of the complete department names can save a substantial amount of data storage space.



Employees (partial listing)

LastName	FirstName	Dept
Mills	Dorothy E.	FOP
Bower	Eileen A.	FIT
Reading	Tony R.	HRS
Judd	Carol A.	HRS
Wonsild	Hanna	AOP
Anderson	Christopher	SMK
Massengill	Annette W.	FOP
Badine	David	COP



DeptCodes

Dept	Department
FOP	Airport Operations
FIT	Corporate Operations
HRS	Finance & IT
AOP	Flight Operations
SMK	Human Resources
COP	Sales & Marketing

Note You can learn about table lookup operations in "Combining Data Horizontally" on page 534.

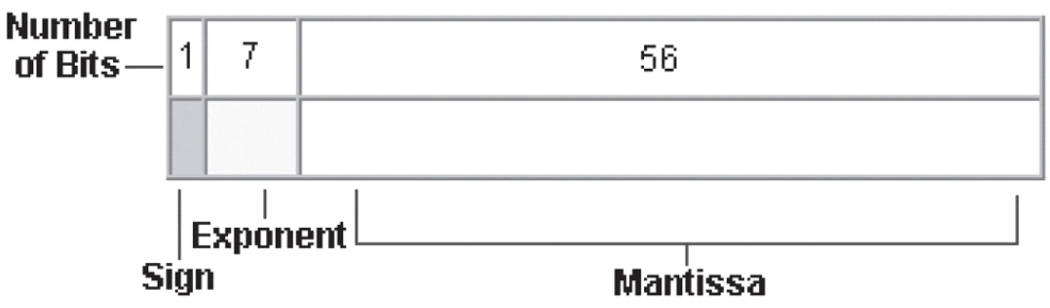
Reducing Data Storage Space for Numeric Variables

Another way to eliminate wasted space and thereby to reduce the amount of data storage space that you need is to reduce the length of *numeric* variables. In addition to conserving data storage space, reduced-length numeric variables use *less I/O*, both when data is written and when it is read. For a file that is read frequently, this savings can be significant. However, in order to safely reduce the length of numeric variables, you need to understand how SAS stores numeric data.

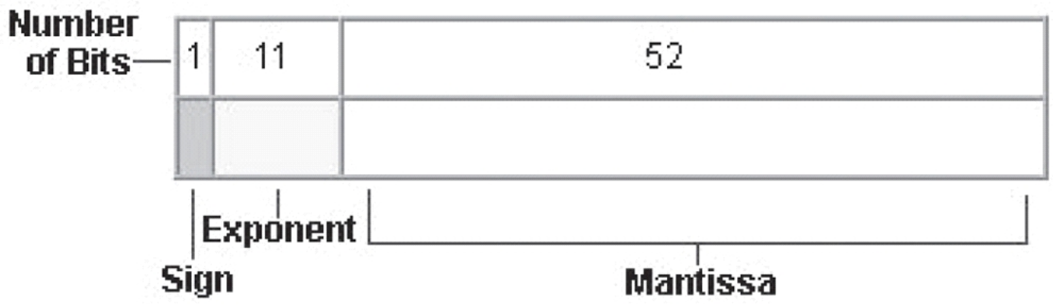
How SAS Stores Numeric Variables

To store numbers of large magnitude and to perform computations that require many digits of precision to the right of the decimal point, SAS stores all numeric values using double-precision floating-point representation. SAS stores the value of a numeric variable as multiple digits per byte. A SAS numeric variable can be from 2 to 8 bytes or 3 to 8 bytes in length, depending on your operating environment. The default length for a numeric variable is 8 bytes.

Floating-point representation is an implementation of a form of scientific notation. For example, the number 234 would be written as **`.234*10**3`** with a base of **`10.`** In this example, **`.234`** is referred to as the mantissa, **`10`** is the base, and **`3`** is the exponent. The figures below show how SAS stores a numeric value in 8 bytes. For mainframe environments, the first bit stores the sign, the next seven bits store the exponent of the value, and the remaining 56 bits store the mantissa.



For non-mainframe environments, the first bit stores the sign, the next eleven bits store the exponent of the value, and the remaining 52 bits store the mantissa.



Note The minimum length for a numeric variable is 2 bytes in mainframe environments and 3 bytes in non-mainframe environments.

Now that you have seen how SAS stores numeric variables, consider how you can assign a length to your numeric variables that is less than the default length of 8 bytes.

Assigning Lengths to Numeric Variables

You can use a *LENGTH statement* to assign a length from 2 to 8 bytes to numeric variables. Remember, the minimum length of numeric variables depends on the operating environment. Also, keep in mind that the LENGTH statement affects the length of a numeric variable only in the output data set. Numeric variables *always* have a length of 8 bytes in the program data vector and during processing.

General form, LENGTH statement for numeric variables:

LENGTH *variable(s) length* <DEFAULT=n>;

where

variable(s)

specifies the name of one or more numeric SAS variables, separated by spaces.

length

is an integer that specifies the length of the variable(s).

DEFAULT=*n*

this optional argument changes the default number of bytes that SAS uses to store the values of any newly created numeric variables. If you use the DEFAULT= argument, you do not need to list any *variable(s)*.

Note Values between 2 and 8 or between 3 and 8 depending on your operating environment are valid for *n* or *length*.

DEFAULT= applies only to numeric variables that are added to the program data vector after the LENGTH statement is compiled. You would list specific variables in the LENGTH statement along with the DEFAULT= argument only if you wanted those variables to have a length other than the value for DEFAULT= If you list individual variables in the LENGTH statement, you must list an integer length for each of them.

Caution You should assign reduced lengths to numeric variables only if those variables have integer values. Fractional numbers lose precision if truncated. You will learn more about the loss of precision with reduced-length numeric variables on the next section of this chapter.

Example

The following program assigns a length of 4 to the new variable **`Sale_Percent`** in the data set *ReducedSales*. The LENGTH statement in this DATA step does not apply to the variables that are read in from the *Sales* data set; those variables will maintain whatever length they had in *Sales* when they are read into *ReducedSales*.

```
data reducedsales;
   length default=4;
   set sales;
   Sale_Percent=15;
run;
```

Maintaining Precision in Reduced-Length Numeric Variables

There is a limit to the values that you can precisely store in a reduced-length numeric variable. You have learned that reducing the number of bytes that are used for storing a numeric variable does *not* affect how the numbers are stored in the program data vector. Instead, specifying a value of less than 8 in the LENGTH statement causes the number to be truncated to the specified length when the value is written to the SAS data set.

You should never use the LENGTH statement to reduce the length of your numeric variables if the values are not integers. Fractional numbers lose precision if truncated. Even if the values are integers, you should keep in mind that reducing the length of a numeric variable limits the integer values that can accurately be stored as a value.

The following table lists the possible storage length for integer values on UNIX or Windows operating environments.

Table 21.1: UNIX/Windows

Length (bytes)	Largest Integer Represented Exactly
3	8,192
4	2,097,152
5	536,870,912
6	137,438,953,472
7	35,184,372,088,832
8	9,007,199,254,740,992

The following table lists the possible storage length for integer values on the z/OS operating environment.

Table 21.2: z/OS

Length (bytes)	Largest Integer Represented Exactly
2	256
3	65,536
4	16,777,216
5	4,294,967,296
6	1,099,511,627,776
7	281,474,946,710,656
8	72,057,594,037,927,936

When you store an integer that is equal to or less than the number listed above as the largest integer that can be represented exactly in a reduced-length variable, SAS truncates bytes that contain only zeros. If the integer that is stored in a reduced-length variable is larger than the recommended limit, SAS truncates bytes that contain numbers other than zero, and the integer value is changed. Similarly, you should not reduce the stored size of non-integer data because it can result in a loss of precision due to the truncation of nonzero bytes.

If you decide to reduce the length of your numeric variables, you might want to verify that you have not lost any precision in your values. Here is one way to do this action.

Using PROC COMPARE

You can use PROC COMPARE to gauge the precision of the values that are stored in a shortened numeric variable by comparing the original variable with the shortened variable. The COMPARE procedure compares the contents of two SAS data sets, selected variables in different data sets, or variables within the same data set.

General form, PROC COMPARE step to compare two data sets:

```
PROC COMPARE BASE=SAS-data-set-one
                compare=SAS-data-set-two;
RUN;
```

where

SAS-data-set-one and *SAS-data-set-two*
specify the two SAS data sets that you want to compare.

PROC COMPARE is a good technique to use for gauging the loss of precision in shortened numeric variables because it shows you whether there are differences in the stored numeric values even if these differences do not show up once the numeric variables have been formatted. PROC COMPARE looks at the two data sets and compares their

- data set attributes
- variables
- variable attributes for matching variables
- observations
- values in matching variables.

Output from the COMPARE procedure includes

- a data set summary
- a variables summary
- a listing of common variables that have different attributes
- an observation summary
- a values comparison summary
- a listing of variables that have unequal values
- a detailed list of value comparison results for variables.

Example

The data set *Company.Discount* contains data about sale dates and discounts for certain retail products. There are 35 observations in *Company.Discount*, which is described below.

Variable	Type	Length	Description
Product_ID	num	8	product ID number
Start_Date	num	4	start date of sale
End_Date	num	5	end date of sale
Unit_Sales_Price	num	8	discounted sales price per unit
Discount	num	8	discount as percent of nonnal sales price

Suppose you shorten the length of the numeric variable **`Discount`.** The DATA step below creates a new data set named *Company.Discount_Short*, whose only difference from *Company.Discount* is that the length of the variable **`Discount`** is 4 instead of 8.

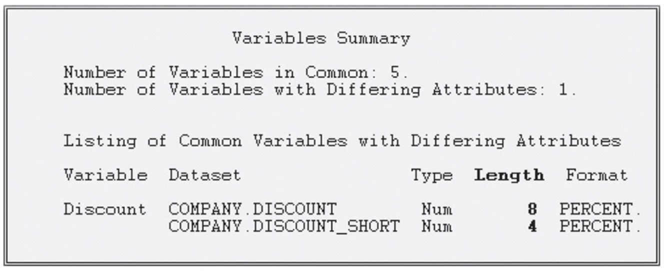
```
data company.discount_short;
   length Discount 4;
   set Company.Discount;
run;
```

You can use PROC COMPARE to evaluate whether shortening the length of **`Discount`** affects the precision of its values by comparing *Company.Discount* to *Company.Discount_Short*.

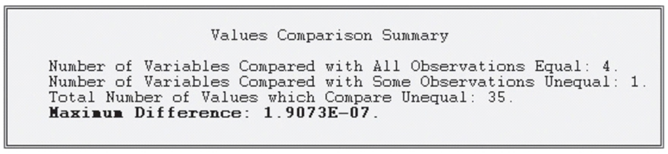
```
proc compare base=company.discount
             compare=company.discount_short;
run;
```

If you were to print these two data sets (*Company.Discount* and *Company.Discount_Short)*, the values might appear to be identical. However, there are differences in the values as they are stored that are not apparent in the formatted output.

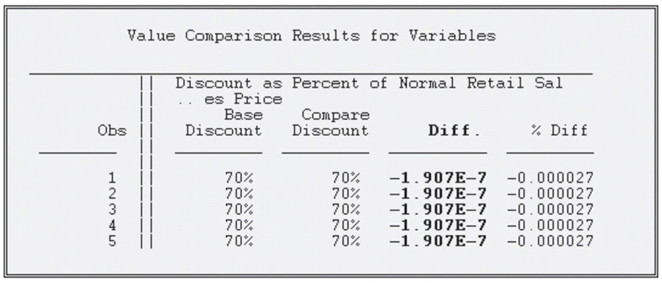
In the partial output below, you can see that shortening the length of **`Discount`** results in a loss of precision in its values; the values for **`Discount`** in *Company.Discount_Short* differ by a maximum of *1.9073E-07*. The value comparison results show that although the values for **`Discount`** in the first five observations appear as *70%* in both data sets, the precise (unformatted) values differ by *—1.907E-7*.



```
                  Variables Summary

   Number of Variables in Common: 5.
   Number of Variables with Differing Attributes: 1.


   Listing of Common Variables with Differing Attributes

   Variable  Dataset                 Type  Length  Format

   Discount  COMPANY.DISCOUNT        Num        8  PERCENT.
             COMPANY.DISCOUNT_SHORT  Num        4  PERCENT.
```



```
                 Values Comparison Summary

  Number of Variables Compared with All Observations Equal: 4.
  Number of Variables Compared with Some Observations Unequal: 1.
  Total Number of Values which Compare Unequal: 35.
  Maximum Difference: 1.9073E-07.
```



```
           Value Comparison Results for Variables

     __________________________________________________________
               ||  Discount as Percent of Normal Retail Sal
               ||  .. es Price
               ||      Base     Compare
          Obs  ||   Discount   Discount       Diff.     % Diff
      ________ ||  _________  _________  _________  _________
               ||
            1  ||       70%        70%   -1.907E-7  -0.000027
            2  ||       70%        70%   -1.907E-7  -0.000027
            3  ||       70%        70%   -1.907E-7  -0.000027
            4  ||       70%        70%   -1.907E-7  -0.000027
            5  ||       70%        70%   -1.907E-7  -0.000027
```

Figure 21.1: Partial PROC COMPARE Output

Comparative Example: Creating a SAS Data Set That Contains Reduced-Length Numeric Variables

Default Versus Reduced-Length Numeric Variables

Suppose you want to create a SAS data set in which to store retail data about a group of orders. Suppose that the data you want to include in your data set is all numeric data and that it is currently stored in a raw data file. You can create the data set using

1. Default-Length Numeric Variables
2. Reduced-Length Numeric Variables

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for creating reduced-length numeric variables.

Note Throughout this book, the keyword _NULL_ is often used as the data set name in sample programs. Using _NULL_ causes SAS to execute the DATA step as if it were creating a new data set. However, no observations or variables are written to an output data set. Using _NULL_ when benchmarking enables you to determine what resources are used to read a SAS data set.

Programming Techniques

---

❶ Default-Length Numeric Variables

This program reads the external data file that is referenced by the fileref *flatl* and creates a new data set called *Retail.Longnums* that contains 12 numeric variables. Each of the variables in *Retail.Longnums* has the default storage length of 8 bytes. The second DATA step in this program reads the numeric variables from *Retail.Longnums*.

```
data retail.longnums;
   infile flat1;
   input Customer_ID               12.
         Employee_ID               12.
         Street_ID                 12.
         Order_Date                date9.
         Delivery_Date             date9.
         Order_ID                  12.
         Order Type                comma16.
         Product_ID                12.
         Quantity                  4.
         Total_Retail_Price        dollarl3 .2
         CostPrice_Per_Unit        dollarl3 .2
         Discount                  5.          ;
run;

data _null_;
   set retail.longnums;
run;
```

❷ Reduced-Length Numeric Variables

This program reads the external data file that is referenced by the fileref *flat1* and creates a new SAS data set called *Retail.Shortnums* that contains 12 numeric variables. A LENGTH statement is used to reduce the storage length of most of the numeric variables in *Retail.Shortnums*, as follows:

- **`Total_Retail_Price`** and **`CostPrice_Per_Unit`** have a storage length of 8 bytes.
- **`Product_ID`** has a storage length of 7 bytes.
- **`Street_ID`** and **`Order_ID`** have a storage length of 6 bytes.
- **`Employee_ID`** has a storage length of 5 bytes.
- **`Customer_ID, Order_Date, Delivery_Date`** and **`Discount`** have a storage length of 4 bytes.
- **`Order_Type`** and **`Quantity`** have a storage length of 3 bytes.

The second DATA step reads the reduced-length numeric variables from *Retail.Shortnums*.

```
data retail.shortnums;
   infile flat1;
          length Quantity Order_Type  3
          Customer_IDOrder_Date
          Delivery_Date Discount      4
          Employee_ID                 5
          Street_IDOrder_ID         6
          Product_ID                  7
          Total_Retail_Price
          CostPrice_Per_Unit         8;
   input  Customer_ID         12.
          Employee_ID         12.
          Street_ID           12.
          Order_Date          date9.
          Delivery_Date       date9.
          Order_ID            12.
          Order_Type          commal6.
          Product_ID          12.
          Quantity            4.
          Total_Retail_Price  dollar13.2
          CostPrice_Per_Unit  dollar13.2
          Discount            5.         ;
run;

data _null_;
   set retail.shortnums;
run;
```

Note Remember that when you reduce the storage length of numeric variables, you risk losing precision in their values. You can use PROC COMPARE to verify the precision of shortened numeric variables.

```
proc compare base=retail.longnums;
             compare=retail.shortnums;
run;
```

General Recommendations

Create reduced-length numeric variables for integer values when you need to conserve data storage space.

Compressing Data Files

Overview

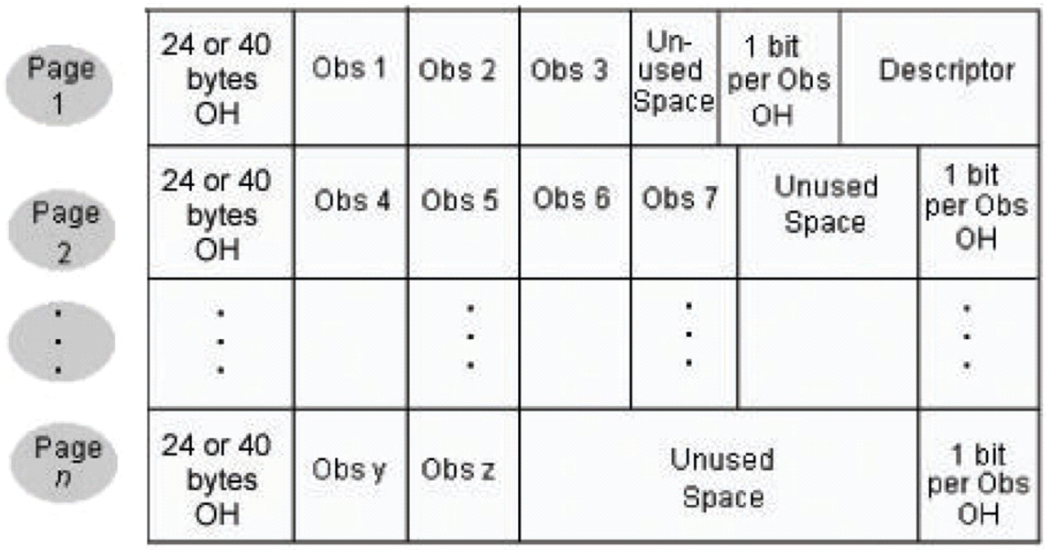
By default, a SAS data file is uncompressed. You can compress your data files in order to conserve disk space, although some files are not good candidates for compression. The file structure of a compressed data file is different from the structure of an uncompressed file. You use the COMPRESS= data set option or system option to compress a data file. You use the POINTOBS= data set option to enable SAS to access observations in compressed files directly rather than sequentially. You use the REUSE= data set option or system option to specify that SAS should reuse space in a compressed file when observations are deleted or updated.

Review of Uncompressed Data File Structure

By default, a SAS data file is not compressed. In uncompressed data files,

- each data value of a particular variable occupies the same number of bytes as any other data value of that variable.
- each observation occupies the same number of bytes as any other observation.
- character values are padded with blanks.
- numeric values are padded with binary zeros.
- there is a 24-byte overhead at the beginning of each page in a 32-bit operating environment (40-byte overhead in a 64-bit operating environment).
- there is a 1-bit per observation overhead (rounded up to the nearest byte) at the end of each page; this bit denotes an observation's status as deleted or not deleted.
- new observations are added at the end of the file. If a new observation will not fit on the current last page of the file, a whole new data set page is added.
- the descriptor portion of the data file is stored at the end of the first page of the file.

The figure below depicts the structure of an uncompressed data file.



Note In a 64-bit operating environment, each page has a 40-byte overhead. In a 32-bit operating environment, each page has a 24-byte overhead.

In comparison, look at the characteristics of a compressed data file.

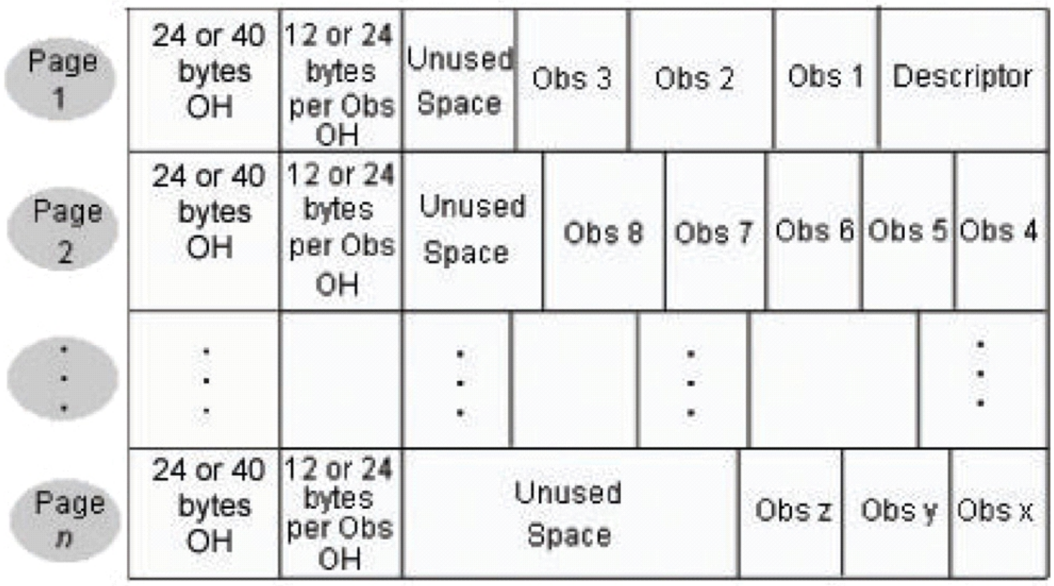
Compressed Data File Structure

Compressed data files

- treat an observation as a single string of bytes by ignoring variable types and boundaries.
- collapse consecutive repeating characters and numbers into fewer bytes.
- contain a 24-byte overhead at the beginning of each page in a 32-bit operating environment or 40-byte overhead in a 64-bit operating environment.
- contain a 12-byte-or 24-byte-per-observation overhead following the page overhead. This space is used for deletion status, compressed length, pointers, and flags.

Each observation in a compressed data file can have a different length, which means that some pages in the data file can store more observations than others can. When an updated observation is larger than its original size, it is stored on the same data file page and uses available space. If not enough space is available on the original page, the observation is stored on the next page that has enough space, and a pointer is stored on the original page.

The image below depicts the structure of a compressed data file.



Note The overhead for each observation is 12 bytes per observation on 32-bit operating environments and 24 bytes per observation on 64-bit operating environments. The overhead for each page is 24 bytes in a 32-bit operating environment and 40 bytes in 64-bit operating environment.

Deciding Whether to Compress a Data File

Not all data files are good candidates for compression. Remember that in order for SAS to read a compressed file, each observation must be uncompressed. This requires more CPU resources than reading an uncompressed file. However, compression can be beneficial when the data file has one or more of the following properties:

- It is large.
- It contains many long character values.
- It contains many values that have repeated characters or binary zeros.
- It contains many missing values.
- It contains repeated values in variables that are physically stored next to one another.

In character data, the most frequently encountered repeated value is the blank. Long text fields, such as comments and addresses, often contain repeated blanks. Likewise, binary zeros are used to pad numeric values that can be stored in fewer bytes than are available in a particular numeric variable. This happens most often when you assign a small or medium-sized integer to an 8-byte numeric variable.

Note If saving disk space is crucial, consider storing missing data as a small integer, such as 0 or 9, rather than as a SAS missing value. Small integers can be compressed more than SAS missing values can.

A data file is *not* a good candidate for compression if it has

- few repeated characters
- small physical size
- few missing values
- short text strings.

Next, look at how to compress a data file.

The COMPRESS= System Option and the COMPRESS= Data Set Option

To compress a data file, you use either the COMPRESS= data set option or the COMPRESS= system option. You use the

COMPRESS= system option to compress all data files that you create during a SAS session. Similarly, you use the *COMPRESS= data set option* to compress an individual data file.

General form, COMPRESS= system option:

OPTIONS COMPRESS= NO | YES | CHAR | BINARY;

where

NO

: is the default setting, which does not compress the data set.

CHAR or YES

: uses the Run Length Encoding (RLE) compression algorithm, which compresses repeating consecutive bytes such as trailing blanks or repeated zeros.

BINARY

: uses Ross Data Compression (RDC), which combines run-length encoding and sliding-window compression.

Caution If you set the COMPRESS= system option to a value other than *NO*, SAS compresses every data set that is created during the current SAS session, including temporary data sets in the *Work* library. Although this might conserve data storage space, it will also use greater amounts of other resources.

General form, COMPRESS= data set option:

DATA *SAS-data-set* **(COMPRESS= NO | YES | CHAR | BINARY);**

where

SAS-data-set

: specifies the data set that you want to compress.

NO

: is the default setting, which does not compress the data set.

CHAR or YES

: uses the Run Length Encoding (RLE) compression algorithm, which compresses repeating consecutive bytes such as trailing blanks or repeated zeros.

BINARY

: uses Ross Data Compression (RDC), which combines run-length encoding and sliding-window compression.

Note The COMPRESS= data set option overrides the COMPRESS= system option.

The *YES* or *CHAR* setting for the COMPRESS= option uses the RLE compression algorithm. RLE compresses observations by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations. Therefore, RLE is most often useful for character data that contains repeated blanks. The *YES* or *CHAR* setting is also good for compressing numeric data in which most of the values are zero.

The *BINARY* setting for the COMPRESS= option uses RDC, which combines run-length encoding and sliding-window compression. This method is highly effective for compressing medium to large blocks of binary data (numeric variables).

A file that has been compressed using the *BINARY* setting of the COMPRESS= option takes significantly more CPU time to uncompress than a file that was compressed with the *YES* or *CHAR* setting. *BINARY* is more efficient with observations that are several hundred bytes or more in length. *BINARY* can also be very effective with character data that contains patterns rather than simple repetitions.

When you create a compressed data file, SAS compares the size of the compressed file to the size of the uncompressed file of the same page size and record count. Then SAS writes a note to the log indicating the percentage of reduction that is obtained by compressing the file.

When you use either of the COMPRESS= options, SAS calculates the size of the overhead that is introduced by compression as well as the maximum size of an observation in the data set that you are attempting to compress. If the maximum size of the observation is smaller than the overhead that is introduced by compression, SAS disables compression, creates an uncompressed data set, and issues a warning message stating that the file was not compressed.

Once a file is compressed, the setting is a permanent attribute of the file. In order to change the setting to uncompressed, you must re-create the file.

Caution Compression of observations is not supported by all SAS engines. See the SAS documentation for the COMPRESS= data set option for more information.

Example

The data set *Company.Customer* contains demographic information about a retail company's customers. The data set includes character variables such as **`Customer_Name, Customer_FirstName, Customer_LastName`**, and **`Customer_Address.`** These character variables have the potential to contain many repeated blanks in their values. The following program will create a compressed data set named *Company.Customers_Compressed* from *Company.Customer* even if the COMPRESS= system option is set to *NO*.

```
data company.customer_compressed (compress=char);
   set company.customer;
run;
```

SAS writes a note to the SAS log about the compression of the new data set, as shown below.

Table 21.3: SAS Log

```
NOTE: There were 89954 observations read from the data
      set COMPANY.CUSTOMER.
NOTE: The data set COMPANY.CUSTOMER_COMPRESSED has 89954
      observations and 11 variables.
NOTE: Compressing data set COMPANY.CUST0MER_C0MPRESSED
      decreased size by 32.81 percent.
      Compressed is 991 pages; un-compressed would require
      1475 pages.
NOTE: DATA statement used (Total process time):
      real time            3.90 seconds
      cpu time             0.96 seconds
```

Now that you have seen how to create a compressed data set, look at working with compressed data sets. In general, you use a compressed data set in your programs in the same way that you would use an uncompressed data set. However, there are two options that relate specifically to compressed data sets.

Accessing Observations Directly in a Compressed Data Set

By default, the DATA step processes observations in a SAS data set sequentially. However, sometimes you might want to access observations directly rather than sequentially because doing so can conserve resources such as CPU time, I/O, and real time. You can use the POINT= option in the MODIFY or SET statement to access observations directly rather than sequentially. You can review information about the POINT= option in "Creating Samples and Indexes" on page 470. You can also use the FSEDIT procedure to access observations directly.

Allowing direct access to observations in a compressed data set increases the CPU time that is required for creating or updating the data set. You can set an option that does not allow direct access for compressed data sets. If it is not important for you to be able to point directly to an observation by number within a compressed data set, it is a good idea to disallow direct access in order to improve the efficiency of creating and updating the data set. Look at how to disallow direct access to observations in a compressed data set.

The POINTOBS= Data Set Option

When you are working with compressed data sets, you use the **POINTOBS=** data set option to control whether observations can be processed with direct access (by observation number) rather than with sequential access only.

General form, POINTOBS= data set option:

DATA *SAS-data-set* **(POINTOBS= YES | NO);**

where

SAS-data-set
: specifies the data set that you want to compress.

YES
: is the default setting, which allows random access to the data set.

NO
: does not allow random access to the data set.

Note In order to use the POINTOBS= data set option, the COMPRESS= option must have a value of *YES, CHAR*, or *BINARY* for the *SAS-data set* that is specified.

Allowing random access to a data set does not affect the efficiency of retrieving information from a data set, but it does increase the CPU usage by approximately 10% when you create or update a compressed data set. That is, allowing random access reduces the efficiency of writing to a compressed data set but does not affect the efficiency of reading from a compressed data set. Therefore, if you do not need to access data by observation number, then by specifying POINTOBS=*NO*, you can improve performance by approximately 10% when creating a compressed data set and when updating or adding observations to it.

Example

The following program creates a data set named *Company.Customer_Compressed* from the *Company.Customer* data set and ensures that random access to the compressed data set is not allowed.

```
data company.customer_compressed (compress=yes pointobs=no);
   set company.customer;
run;
```

Now look at an option that enables you to further reduce the data storage space that is required for your compressed data sets.

The REUSE= System Option and the REUSE= Data Set Option

In a compressed data set, SAS appends new observations to the end of the data set by default. If you delete an observation within the data set, empty disk space remains in its place. However, it is possible to track and reuse free space within the data set when you delete or update observations. By reusing space within a data set, you can conserve data storage space.

The *REUSE= system option* and the *REUSE= data set option* specify whether SAS reuses space when observations are added to a compressed data set. If you set the REUSE= data set option to *YES* in a DATA statement, SAS tracks and reuses space in the compressed data set that is created in that DATA step. If you set the REUSE= system option to *YES*, SAS tracks and reuses free space in all compressed data sets that are created for the remainder of the current SAS session.

General form, REUSE= system option:

OPTIONS REUSE= NO | YES;

where

NO

is the default setting, which specifies that SAS does not track unused space in the compressed data set.

YES

specifies that SAS tracks free space and reuses it whenever observations are added to an existing compressed data set.

General form, REUSE= data set option:

DATA *SAS-data-set* **(COMPRESS=YES REUSE=NO | YES);**

where

SAS-data-set

specifies the data set that you want to compress.

NO

is the default setting, which specifies that SAS does not track unused space in the compressed data set.

YES

specifies that SAS tracks free space and reuses it whenever observations are added to an existing compressed data set.

Note The REUSE= data set option overrides the REUSE= system option.

If the REUSE= option is set to *YES*, observations that are added to the SAS data set are inserted wherever enough free space exists, instead of at the end of the SAS data set.

Specifying *NO* for the REUSE= option results in less efficient usage of space if you delete or update many observations in a SAS data set because there will be unused space within the data set. With the REUSE= option set to *NO*, the APPEND procedure, the FSEDIT procedure, and other procedures that add observations to the SAS data set add observations to the end of the data set, as they do for uncompressed data sets.

You cannot change the REUSE= attribute of a compressed data set after it is created. This means that space is tracked and reused in the compressed SAS data set according to the value of the REUSE= option that was specified when the SAS data set was created, not when you add and delete observations. Also, you should be aware that even with the REUSE= option set to *YES*, the APPEND procedure will add observations to the end of the data set.

Caution Specifying *YES* as the value for the REUSE= option causes the POINTOBS= option to have a value of AO even if you specify *YES* as the value for POINTOBS=. The insertion of anew observation into unused space (rather than at the end of the data set) and the use of direct access are not compatible.

Example

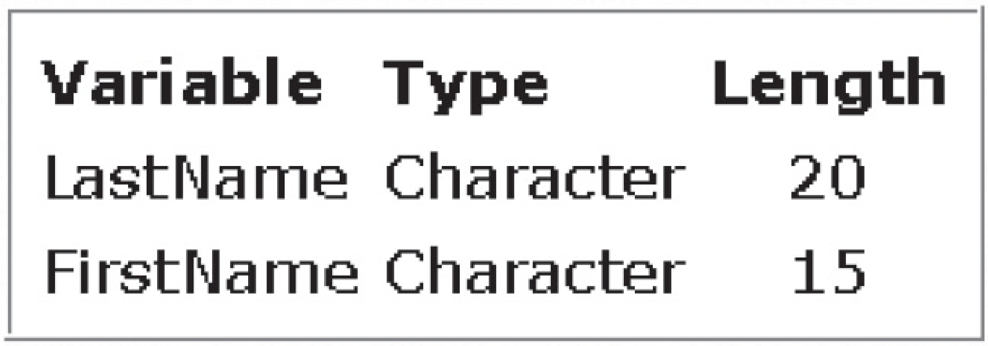
The following program creates a compressed data set named

Company.Customer_Compressed from the *Company.Customer* data set. Because the REUSE= option is set to *YES*, SAS will track and reuse any empty space within the compressed data set.

```
data company.customer_compressed (compress=yes reuse=yes);
   set company.customer;
run;
```

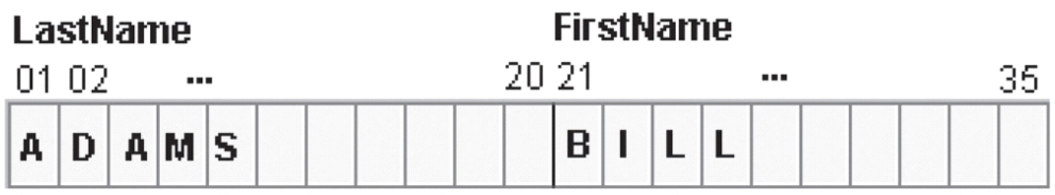
How SAS Compresses Data

Look at how SAS compresses data. A fictional data set named *Roster* is described in the table below.

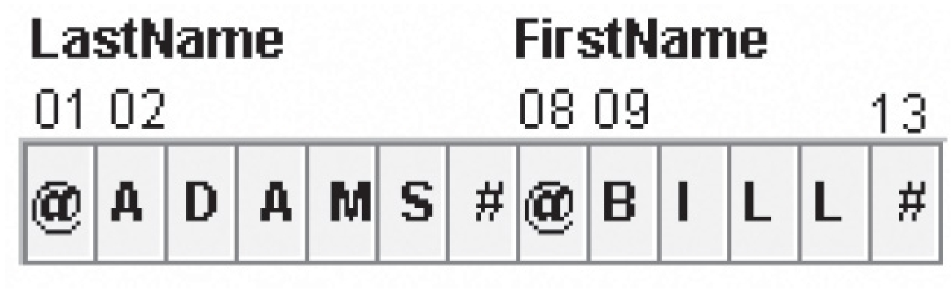


Variable	Type	Length
LastName	Character	20
FirstName	Character	15

In uncompressed form, each observation in *Roster* uses a total of 35 bytes to store these two variables: 20 bytes for the first variable, **`LastName`**, and 15 bytes for the second variable, **`FirstName.`** The image below illustrates the storage of the first observation in the uncompressed version of *Roster*.



Suppose that you use the *CHAR* setting for the COMPRESS= option to compress *Roster*. In compressed form, the repeated blanks are removed from each value. The first observation from *Roster* uses a total of only 13 bytes: 7 for the first variable, **`LastName`**, and 6 for the second variable, **`FirstName.`** The image below illustrates the storage of the first observation in the compressed version of *Roster*.



The @ indicates the number of uncompressed characters that follow. The # indicates the number of blanks repeated at this point in the observation. Only a SAS engine can access these bytes. You cannot print or manipulate them.

Ross Data Compression (COMPRESS=BINARY) uses both run-length encoding and sliding window compression. Suppose a SAS data set has these variables:

Name	Type	Length
Answerl	Num	8
Answer2	Num	8
…		
Answer200	Num	8

In uncompressed form, the SAS data file resembles this:

```
1     2     3     4     5     6     7     8    9
@     +/1   1     #     @     +/1   2     #     %
```

The `@` symbol indicates how many uncompressed characters follow. In the file, **`+/1`** is the sign and exponent. The **`#`** indicates the number of binary zeros that were removed. The `%` represents how many times these values are repeated.

Note Remember that in a compressed data set, observations might not all have the same length, because the length of an observation depends on the length of each value in the observation.

Comparative Example: Creating and Reading Compressed Data Files

Overview

Suppose you want to create two SAS data sets from data that is stored in two raw data files. The raw data file that is referenced by the fileref***fiat1*** contains numeric data about customer orders for a retail company; you want to create a SAS data set named *Retail.Orders* from this raw data file. The raw data file that is referenced by the fileref *flat2* contains character data about customers for a retail company; you want to create a SAS data set named *Retail.Customers* from this raw data file.

In both cases, you can use the DATA step to create either an uncompressed data file or a compressed data file. Furthermore, you can use either binary or character compression in either case. That is, you can use the following techniques:

1. Numeric Data, No Compression
2. Numeric Data, BINARY Compression
3. Numeric Data, CHAR Compression
4. Character Data, No Compression
5. Character Data, BINARY Compression
6. Character Data, CHAR Compression.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for creating compressed data tiles.

Programming Techniques

---

❶ Numeric Data, No Compression

```
data retail.orders (compress=no);
   infile flat1;
   input Customer_ID           12.
         Employee_ID           12.
         Street_ID             12.
         Order_Date            date9.
         Delivery_Date         date9.
         Order_ID              12.
         Order_Type            comma16.
         Product_ID            12.
         Quantity              4.
         Total_Retail_Price    dollar13.
         CostPrice_Per_Onit    dollar13.
         Discount              5.       ;
run;

data _null_;
   set retail.orders;
run;
```

❷ Numeric Data, BINARY Compression

The following program creates the SAS data set *Retail.Orders_binary*, which contains numeric data and uses *BINARY* compression. The second DATA step reads the compressed data file.

```
data retail.orders_binar} (compress=binary);
   infile flat1;
   input  Customer_ID                 12.
          Employee_ID                 12.
          Street_ID                   12.
          Order_Date                  date9.
          Delivery_Date               date9.
          Order_ID                    12.
          Order_Type                  comma16.
          Product_ID                  12.
          Quantity                    4.
          Total_Retail_Price          dollar13.
          CostPrice_Per_Unit          dollar13.
          Discount                    5.       ;
run;

data _null_;
   set retail .orders_binary;
run;
```

❸ Numeric Data, CHAR Compression

The following program creates the SAS data set *Retail.Orders_char*, which contains numeric data and uses *CHAR* compression. The second DATA step reads the compressed data file.

```
data retail.orders_cha](compress=char);
   infile flat1;
   input  Customer_ID              12.
          Employee_ID              12.
          Street_ID                12.
          Order_Date               date9.
          Delivery_Date            date9.
          Order_ID                 12.
          Order_Type               comma16.
          Product_ID               12.
          Quantity                 4.
          Total_Retail_Price       dollar13.
          CostPrice_Per_Unit       dollar13.
          Discount                 5.
run;

data _null_;
   set retail.orders_char;
run;
```

❹ Character Data, No Compression

The following program creates the SAS data set *Retail.Customers*, which contains character data and is uncompressed. The second DATA step reads the uncompressed data file.

```
data retail.customers (compress=no);
   infile flat2;
   input  Customer_Country          $40.
          Customer_Gender           $1.
          Customer_Name             $40.
          Customer_FirstName        $20.
          Customer_LastName         $30.
          Customer_Age_Group        $12.
          Customer_Type             $40.
          Customer_Group            $40.
          Customer_Address          $45.
          Street_Number             $8.          ;
run;

data _null_;
   set retail.cutomers;
run;
```

❺ Character Data, BINARY Compression

The following program creates the SAS data set *Retail.Customers_binary*, which contains character data and uses

BINARY compression. The second DATA step reads the compressed data file.

```
data retail.customers_binar}(compress=binary);
   infile flat2;
   input  Customer_Country         $40
          Customer_Gender          $1.
          Customer_Name            $40
          Customer_FirstName       $20
          Customer_LastName        $30
          Customer_Age_Group       $12
          Customer_Type            $40
          Customer_Group           $40
          Customer_Address         $45.
          Street_Number            $8.
run;

data _null_;
   set retail.customers_binary;
run;
```

6 Character Data, CHAR Compresision

The following program creates the SAS data set *Retail.Customers_char*, which contains character data and uses *CHAR* compression. The second DATA step reads the compressed data file.

```
data retail.customers_cha](compress=char);
   infile flat2;
   input  Customer_Country     $40.
          Customer_Gender      $1.
          Customer_Name        $40.
          Customer_FirstName   $20.
          Customer_LastName    $30.
          Customer_Age_Group   $12.
          Customer_Type        $40.
          Customer_Group       $40.
          Customer_Address     $45.
          Street_Number        $8.      ;
run;

data _null_;
   set retail.customers_char;
run;
```

General Recommendations

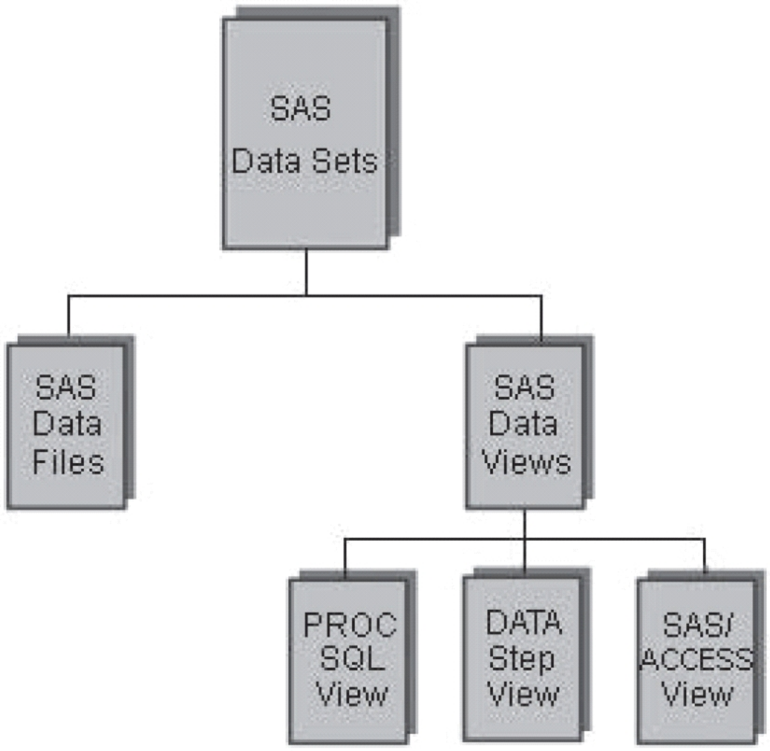
- Save data storage space by compressing data, but remember that compressed data causes an increase in CPU usage because the data must be uncompressed for processing. Compressing data always uses more CPU resources than not compressing data.
- Use binary compression only if the observation length is several hundred bytes or more.

Using SAS DATA Step Views to Conserve Data Storage Space

Overview

Another way to save disk space is to leave your data in its original location and use a SAS data view to access it. Before looking at working with data views, look at what a SAS data view is and how it compares to a SAS data tile.

A SAS data tile and a SAS data view are both types of SAS data sets. The first type, a SAS data file, contains both descriptor information about the data and the data values. The second type, a SAS data view, contains only descriptor information about the data and instructions on how to retrieve data values that are stored elsewhere.



The main difference between SAS data files and SAS data views is where the data values are stored. A SAS data file contains the data values, and a SAS data view does not contain the values. Therefore, data views can be particularly useful if you are working with data values that change often.

Suppose you have a flat file that you read into a SAS data file. If the values in the flat file change, you need to update the data file in order to reflect those changes so that you access the correct values when you reference the data file. However, suppose you use a data view to access the values in your flat file instead of reading those values into a data file. You do not need to update the data view when the values in your flat file change, because each time you reference the view it will execute and access the most recent values in your flat file.

In most cases, you can use a SAS data view as if it were a SAS data file, although there are a few things to keep in mind when you are working with data views.

Note There are multiple types of SAS data views. This chapter discusses only DATA step views. To learn more about PROC SQL views, see "Creating and Managing Views Using PROC SQL" on page 260. For more information about SAS data views and SAS data files, see the SAS documentation.

Now look at DATA step views.

DATA Step Views

A DATA step view contains a partially compiled DATA step program that can read data from a variety of sources, including

- raw data files
- SAS data files
- PROC SQL views
- SAS/ACCESS views
- DB2, ORACLE, or other DBMS data.

A DATA step view can be created only in a DATA step. A DATA step view cannot contain global statements, host-specific data set options, or most host-specific FILE and INFILE statements. Also, a DATA step view cannot be indexed or compressed.

You can use DATA step views to

- always access the most current data in changing tiles
- avoid storing a copy of a large data file
- combine data from multiple sources.

The compiled code does not take up much room for storage, so you can create DATA step views to conserve disk space. On the other hand, use of DATA step views can increase CPU usage because SAS must execute the stored DATA step program each time you use the view.

To create a DATA step view, specify the VIEW= option after the final data set name in the DATA statement.

General form, DATA step to create a DATA step view:

```
DATA SAS-data-view <SAS-data-file-l… SAS data-file-n> /
       VIEW=SAS-data-view;
    <SAS statements>
RUN;
```

where

SAS-data-view
: names the data view to be created.

SAS-data-file-1…SAS-data-file-n
: is an optional list that names any data files to be created.

SAS statements
: includes other DATA step syntax to create the data view and any data files that are listed in the DATA statement.

The VIEW= option tells SAS to compile, but not to execute, the source program and to store the compiled code in the input DATA step view that is named in the option.

> **Note** If you specify additional data files in the DATA statement, SAS creates these data files when the view is processed in a subsequent DATA or PROC step. Therefore, you need to reference the data view before you attempt to reference the data file in later steps.

Example

The following program creates a DATA step view named *Company.Newdata* that reads from the file referenced by the fileref in the INFILE statement.

```
data company.newdata / view=company.newdata;
   infile <fileref>;
   <DATA step statements>
run;
```

The DESCRIBE Statement

DATA step views retain source statements. You can retrieve these statements by using the DESCRIBE statement. The following example uses the DESCRIBE statement in a DATA step to write a copy of the source code for the data view *Company.Newdata* to the SAS log:

```
data view=company.newdata;
   describe;
run;
```

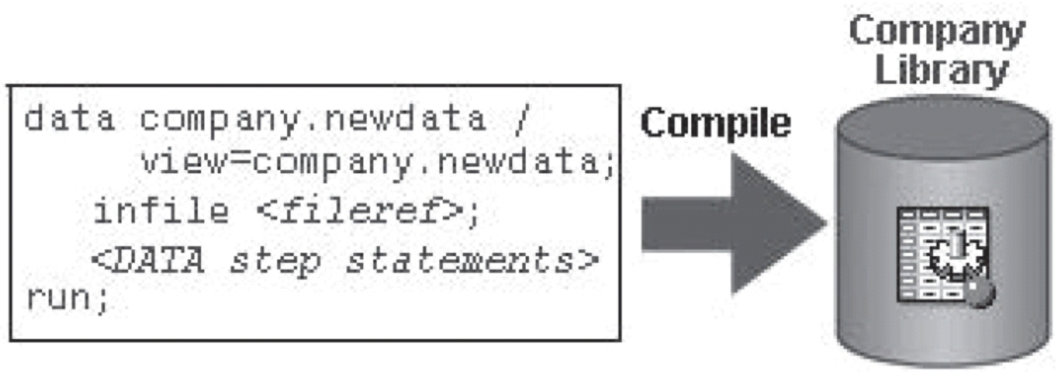
Now look at using DATA step views.

Creating and Referencing a SAS DATA Step View

In order to use DATA step views successfully, you need to understand what happens when you create and reference one.

When you create a DATA step view,

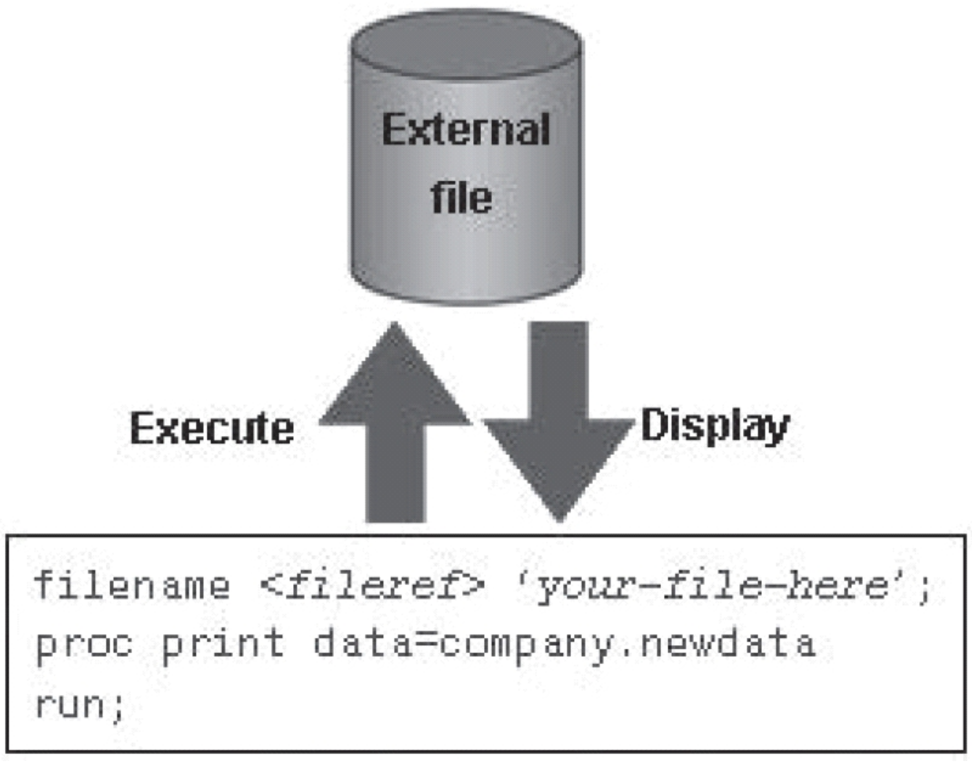
- the DATA step is partially compiled
- the intermediate code is stored in the specified SAS library with a member type of VIEW.



```
data company.newdata /
     view=company.newdata;
   infile <fileref>;
   <DATA step statements>
run;
```

You reference a DATA step view in the same way that you reference a data file. When you reference the view in a subsequent DATA or PROC step,

- the compiler resolves the intermediate code and generates executable code for the host environment
- the generated code is executed as the DATA or PROC step requests observations.



```
filename <fileref> 'your-file-here';
proc print data=company.newdata
run;
```

You can use a DATA step view as you would use any other SAS data set, with the exception that you cannot write to the view except under very specific circumstances. Also, you should keep in mind that a SAS data view reads from its source files each time it is used, so if the data that it is accessing changes, the view will change also. Likewise, if the structure of the data that a view accesses changes, you will probably need to alter the view in order to account for this change.

Note In SAS 9.1, the OBSBUF= data set option enables you to specify how many observations to read at one time from the source data for the DATA step view. The default size of the view buffer is 32K, which means that the number of observations that can be read into the view buffer at one time depends on the observation length.

If the observation length is larger than 32K, then only one observation can be read into the buffer at a time.

Remember that although data views conserve data storage space, processing them can require more resources than processing a data tile. Look at a few situations where using a data view can adversely affect processing efficiency.

Referencing a Data View Multiple Times in One Program

SAS executes a view each time it is referenced, even within one program. Therefore, if data is used many times in one program, it is more efficient to create and reference a temporary SAS data file than to create and reference a view.

Example

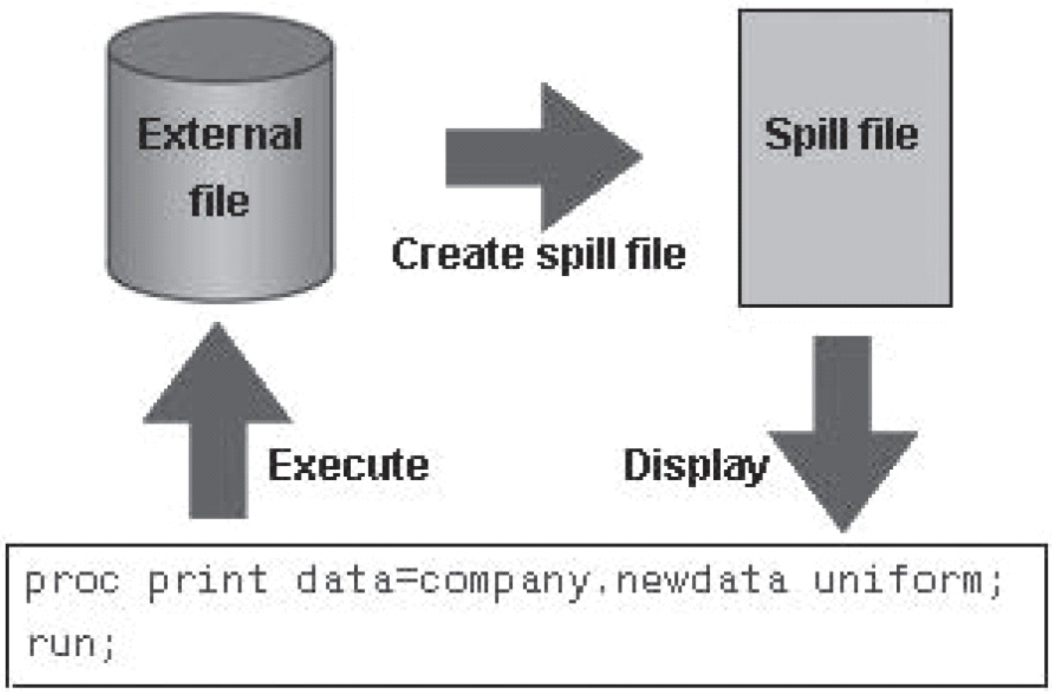
Instead of referencing a data view in each step in the program, you could add a DATA step to the beginning of the program to create a temporary data file and read the data view into it. Then you could reference the temporary data set in each of the subsequent steps. By referencing the temporary data file rather than the data view in each of the PROC steps, you enable SAS to execute the data view only once instead of three times.

There are other reasons why extracting data to a temporary data file is a good idea. Suppose you submit this code and it takes a long time to run. If someone changes the flat file while your code is running, you will have inconsistent results unless you have created a SAS data file before submitting the PROC PRINT, PROC FREQ, and PROC MEANS steps, and you use the data file in your program.

Making Multiple Passes through Data in a Data View

Expect a degradation in performance when you use a SAS data view with a procedure that requires multiple passes through the data. When multiple passes are requested, the view must build a cache, which is referred to as a spill file, that contains all generated observations. Then SAS reads the data in the spill file on each of the multiple passes through the data in order to ensure that subsequent passes read the same data that was read by previous passes.

For example, the UNIFORM option of the PRINT statement makes all the columns consistent from page to page by determining the longest value for a particular variable. In order to do this, SAS must make two passes through the data: one pass to find the longest value in the data, and one pass to print the data. If you use the UNIFORM option to print a data view, SAS creates a spill file as it generates observations from the view. Then SAS makes two passes through the observations that are in the spill file.



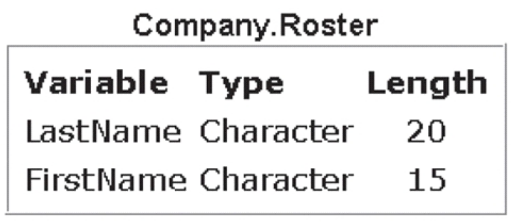
```
proc print data=company.newdata uniform;
run;
```

Note Some statistical procedures pass through the data more than once.

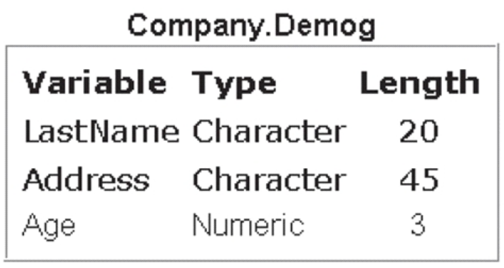
Creating Data Views on Unstable Data

Avoid creating views on files whose structures often change. If the view describes the structure of the raw data file, you need to make changes to the view each time the file changes.

For example, suppose you create a view that combines the data file *Company.Roster* with the data file *Company.Demog*. *Roster* contains the variables **`LastName`** and **`FirstName`**, and *Company.Demog* contains the variables **`LastName, Address`**, and **`Age`**, as shown below.


Company.Roster

Variable	Type	Length
LastName	Character	20
FirstName	Character	15

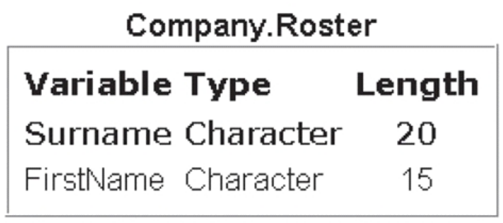

Company.Demog

Variable	Type	Length
LastName	Character	20
Address	Character	45
Age	Numeric	3

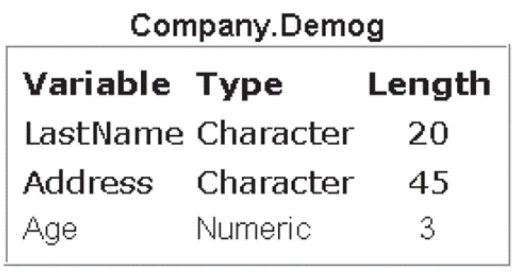
Suppose that both *Company.Roster* and *Company.Demog* are sorted by **`LastName.`** You could use a MERGE statement to combine these two data files into a view named *Company.Roster_View*, as shown below.

```
data company.roster_view/view=company.roster_view;
   merge company.roster company.demog;
   by lastname;
run;
```

Now suppose *Company.Roster* changes so that **`LastName`** is named **`Surname.`** Your data view must also be updated.


Company.Roster

Variable	Type	Length
Surname	Character	20
FirstName	Character	15


Company.Demog

Variable	Type	Length
LastName	Character	20
Address	Character	45
Age	Numeric	3

```
data roster_view/view=roster_view;
   merge company.roster company.demog(rename=(LastName=Surname));
   by lastname;
run;
```

If *Company.Roster* changed again so that **`Surname`** and **`FirstName`** were combined into one variable called **`FullName`**, the code for your data view would need additional changes. Although this is a simple example, you can see that a data view that is based on unstable data will require additional maintenance work.

Comparative Example: Creating and Reading a SAS Data View

Overview

Suppose you have two SAS data sets, *Retail.Custview* and *Retail.Custdata*, that have been created from the same raw data tile. *Retail.Custview* is a DATA step view, and *Retail.Custdata* is a data tile. You can use these two data sets to compare the disk space that is required for each as well as the resources that are used to read from each:

1. Data View
2. Data File

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for using SAS DATA step views.

Programming Techniques

---

❶ Data View

This program reads data from a raw data file, creates a SAS DATA step view named *Retail.Custview*, and then reads from the new DATA step view. The first DATA step creates the data view *Retail.Custview*. The second DATA step reads from the DATA step view.

```
data retail.custview / view = retail.custview;
   infile flat1;
   input @1   Customer_ID         12.
         @13  Country             $2.
         @15  Gender              $1.
         @16  Personal_ID        15.
         @31  Customer_Name      $40.
         @71  Customer_FirstName $20.
         @91  Customer_LastName  $30.
         @121 Birth_Date        date9.
         @130 Customer_Address   $45.
         @175 Street_ID          12.
         @199 Street Number      $8.
         @207 Customer_Type_ID    8.;
run;

data _null_;
   set retail.custview;
run;
```

❷ Data File

This program reads data from a raw data file, creates a SAS data file named *Retail.Custdata*, and reads from the new SAS data file. The first DATA step creates the data file *Retail.Custdata*. The second DATA step reads from the data file.

```
data retail.custdata;
   infile flat1;
   input @1   Customer_ID          12.
         @13  Country              $2.
         @15  Gender               $1.
         @16  Personal_ID          $15.
         @31  Customer_Name        $40.
         @71  Customer FirstName   $20.
         @91  Customer LastName    $30.
         @121 Birth_Date           date9.
         @130 Customer Address     $45.
         @175 Street_ID            12.
         @199 Street_Number        $8.
         @207 Customer_Type_ID     8.;
run;

data _null_;
   set retail.custdata;
run;
```

General Recommendations

- Create a SAS DATA step view to avoid storing a raw data file and a copy of that data in a SAS data file.
- Use a SAS DATA step view if the content, but not the structure, of the flat file is dynamic.
- Create a DATA step view to combine multiple SAS data sets with a merge or concatenate.

- Create a DATA step view to subset the data for frequently used subsets.

Summary

Reducing Data Storage Space for Character Variables

SAS stores character data as one character per byte. The default length for a character variable is 8 bytes. You can use the LENGTH statement to increase or reduce the length of a character variable. You can also use other coding techniques to reduce the space that is needed for storing your character data.

Reducing Data Storage Space for Numeric Variables

SAS stores numeric data in floating-point representation. The default length for a numeric variable is 8 bytes. You can use a LENGTH statement to reduce the length of a numeric variable. Reading reduced-length numeric variables requires less I/O but more CPU resources than reading full-length numeric variables. You should store only integer values in reduced-length numeric variables, and you should limit the values according to the length that you use. You can use PROC COMPARE to see the precision loss, ifany, in the values of reduced-length numeric variables.

Review the related comparative example:

- "Comparative Example: Creating a SAS Data Set That Contains Reduced-Length Numeric Variables" on page 738.

Compressing Data Files

By default, a SAS data tile is uncompressed. You can compress your data tiles in order to conserve disk space, although some tiles are not good candidates. Use the COMPRESS= data set option or system option to compress a data tile. You use the POINTOBS= data set option to enable SAS to access observations in compressed tiles directly rather than sequentially. You use the REUSE= data set option or system option to specify that SAS should reuse space in a compressed tile when observations are deleted or updated.

Review the related comparative example:

- "Comparative Example: Creating and Reading Compressed Data Files" on page 749.

Using SAS DATA Step Views to Conserve Data Storage Space

You can leave your data in its original storage location and use SAS data views to access the data in order to reduce the amount of space needed for storing data on disk. A DATA step view is a specific type of data view that is created in a DATA step with the VIEW= option. You use the DESCRIBE statement to write the source code for a data view to the SAS log. Some of the advantages of using DATA step views rather than data files are that they always access the most recent data in dynamic files and that they require less disk space. However, there can be an effect on performance when you use a DATA step view.

Review the related comparative example:

- "Comparative Example: Creating and Reading a SAS Data View" on page 759.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which of the following statements about uncompressed SAS data files is true? ?

a. The descriptor portion is stored on whatever page has enough room for it.

b. New observations are always added in the first sufficient available space.

c. Deleted observation space is tracked.

d. New observations are always added at the end of the data set.

2. Which of the following statements about compressed SAS data files is true? ?

a. The descriptor portion is stored on whatever data set page has enough room for it.

b. Deleted observation space can be reused.

c. Compressed SAS data files have a smaller overhead than uncompressed SAS data files.

d. In a compressed SAS data set, each observation must be the same size.

3. Which of the following programs correctly creates reduced-length numeric variables? ?

a.
```
a. data temp;
      infile file1;
      input x 4.
            y 3.
            z 2.;
   run;
```

b.
```
b. data temp;
      format x 4.
             y 3.
             z 3.;
      infile file1;
      input x 4.
            y 3.
            z 2.;
   run;
```

c.
```
c. data temp;
      length x 4
             y 3
             z 3;
      infile file1;
      input x 4.
            y 3.
            z 2.;
   run;
```

d.
```
d. data temp;
      informat x 4.
               y 3.
               z 3.;
      infile file1;
      input x 4.
            y 3.
            z 2.;
   run;
```

4. Which of the following statements about SAS data views is true? ?

a. SAS data views use less disk space but more CPU resources than SAS data files.

b. SAS data views can be created only in permanent SAS libraries.

c. SAS data views use less CPU resources but more disk space than SAS data files.

d. SAS data views can be created only in temporary SAS data libraries.

5. Which of the following programs should you use to detect any loss of precision between the default-length numeric variables in *Company.Regular* and the reduced-length numeric variables in the data set *Company.Reduced?* ?

a.
```
a. proc contents data=company.regular;
      compare data=company.reduced;
   run;
```

b.
```
b. proc compare base=company.regular
      compare=company.reduced;
   run;
```

c.
```
c. proc print data=company.regular;
   run;

   proc print data=company.reduced;
   run;
```

d.
```
d. proc datasets library=company;
      contents data=regular compare=reduced;
   run;
```

Answers

1. Correct answer: d

   The descriptor portion of an uncompressed data file is always stored at the end of the first data set page. New observations are always added to the end of the data set, and deleted observation space is neither tracked nor reused.

2. Correct answer: b

   The descriptor portion of a compressed data file is always stored at the end of the first data set page. If you specify *REUSE=YES*, SAS tracks and reuses deleted observation space within a compressed data file. Therefore, every observation in a compressed data file can be a different size. Compressed data files do have a larger overhead than uncompressed data files.

3. Correct answer: c

   You use the LENGTH statement to assign a reduced length to a numeric variable. If you do not use the LENGTH statement to define a reduced length for numeric variables, their default length is 8 bytes. The FORMAT statement associates a format with a variable, and the INFORMAT statement associates an informat with a variable.

4. Correct answer: a

   SAS data views use significantly less disk space than SAS data files. However, SAS data views typically need more CPU resources than SAS data files. You can create a SAS data view in either the temporary SAS data library or in a permanent SAS data library.

5. Correct answer: b

   You use the COMPARE procedure to detect any differences in the values of two data sets. The COMPARE statement is not valid syntax in either the CONTENTS procedure or the DATASETS procedure. Printing both data sets might not reveal differences in the precise values of the shortened variables, depending on the formats that are used.